EasyChair Preprint

№ 4231

A New Advantage Actor-Critic Algorithm For Multi-Agent Environments

Gabor Paczolay and Istvan Harmati

September 21, 2020

A New Advantage Actor-Critic Algorithm For Multi-Agent Environments

Gabor Paczolay
Department of Control Engineering
Budapest University of Technology and Economics
Budapest, Hungary
paczolay.gabor@gmail.com

Istvan Harmati
Department of Control Engineering
Budapest University of Technology and Economics
Budapest, Hungary
harmati@iit.bme.hu

***Abstract*—Reinforcement learning is one of the most researched fields of artificial intelligence right now. Newer and newer algorithms are being developed, especially for deep reinforcement learning, where the selected action is computed with the assist of a neural network. One of the subcategories of reinforcement learning is multi-agent reinforcement learning, where multiple agents are present in the world. In our paper, we modify an already existing algorithm, the Advantage Actor-Critic (A2C) to be suitable for multi-agent scenarios. Afterwards, we test the modified algorithm on our testbed, a cooperative-competitive pursuit-evasion environment.**

***Index Terms*—reinforcement learning, multiagent learning**

I. Introduction

Artificial intelligence is one of the most important fields of today. Of them , reinforcement learning is one of the most researched fields right now. Newer and newer algorithms are being developed to reach successful learning in more situations or to learn the reinforcement learning system with less samples.

In reinforcement learning, a new challenge frontier arises when we take other agents into consideration, this research field is called multi-agent learning. Dealing with other agents - either cooperative(when agents are cooperating with each other) or competitive(when the agents are competing against each other), or a mixture of both - takes learning closer to real-world scenarios, as in real life, no agent acts solely - even random counteracts can be treated as "counteracts of nature".

In our work, we optimize the Synchronous Actor-Critic algorithm to perform better in cooperative multi-agent scenarios.This would enable us more performance when dealing with scenarios where agents help each other.

Littman [1] utilized the Minimax-Q algorithm, a zero-sum multiagent reinforcement learning algorithm first and applied it to a simplified version of a robotics soccer game. Hu and Wellmann [2] created the Nash-Q algorithm and used it on a small gridworld example to show its results. Bowling [3] varied the learning rate of the training process to speed it up while ensuring convergence. Later, he applied the Win or Learn Fast methodology to an actor-critic algorithm to improve its multi-agent capabilities [4].

Reinforcement learning received a huge improvement when neural networks gained popularity and convergence was improved. Mnih et al. [5] applied deep reinforcement learning to playing Atari games successfully by feeding multiple frames at once and by ensuring convergence by utilizing experience replay. Later, deep reinforcement learning was applied to multi-agent systems, for example, to independent multi-agent reinforcement learning. Foerster et al. [6] stabilized experience replay for independent Q-learning by using fingerprints. Omidshafiei et al. [7] utilized Decentralized Hysteretic Deep Recurrent Q-networks for partially observable multi-task multi-agent reinforcement learning problems. Multiple advancements have been made in the filed of centralized learning and decentralized execution as well. Foerster et al. [8] created Counterfactual Multi-Agent Policy Gradients where multi-agent credit assignment was solved. Peng et al. [9] created Multiagent Bidirectionally-Coordinated Nets with Actor-Critic hierarchy and Recurrent Neural Networks for communication. Sunehag et al. [10] utilized Value Decomposition Networks with common reward and Q function decomposition. Rashid et al. [11] utiized QMIX with Value Function Factorization, Q-function decomposition with the help of a feed-forward neural network with better performance than the former value decomposition one. Lowe et al. [12] improved the Deep Deterministic Policy Gradient by a altering the critic to contain all actions of all agents, thus making the algorithm capable of processing more multi-agent scenarios. Shihui et al. [13] improved upon the previous MADDPG algorithm by making it perform better in zero-sum competitive scenarios by utilizing a method based on Minimax-Q learning. Casgrain et al. [14] upgraded the Deep Q-network algorithm by utilizing methods based on Nash equilibria and thus making it capable of solving multi-agent environments.

Some benchmarks have also been created to analyze performance of various algorithms in multi-agent environments. Vinyals et al. [15] modified the Starcraft II game to be a learning environment. Samvelyan et al. [16] also pointed to Starcraft as a multi-agent benchmark, but in a micromanagement way. Liu et al. [17] introduced a multi-agent soccer environment with continuous simulated physics. Bard et al. [18] reached a new frontier with the cooperative Hanabi game benchmark.

In our work, we modify the already existing Advantage Actor-Critic (A2C) Algorithm to better be suitable for multi-agent scenarios by creating a single-critic version of the

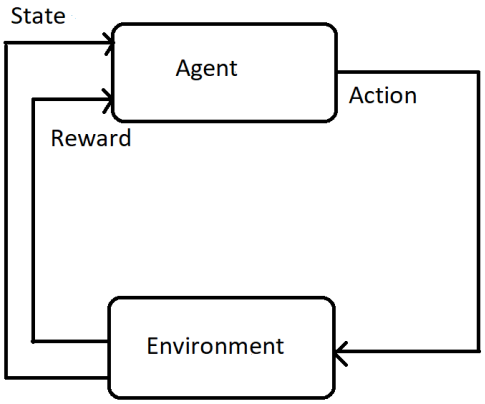

Fig. 1. Markov Decision Process

algorithm. Afterwards, we test this modified A2CM algorithm on our cooperative-competitive pursuit-evasion testbed.

In the following section, we give a theoretical background of our work. Then, the experiments themselves and the testbed is introduced. We continue by showing the results and end with conclusions on the results and suggestions on future work on the topic.

II. Theoretical background

A. Markov Decision Processes

A Markov Decision Process is a mathematical framework for modeling of decision making, as it is shown on Figure 1. In a Markov Decision Process there are states, selectable actions, transition probabilities and rewards. At each timestep the process starts at a state s, and it selects an action a from the available action space. Then, it gets a corresponding reward r, and then finds itself in a state s' given by the probability of $P(s, s')$. A process is said to be Markovian if

$$P(a^t = a|s^t, a^{t-1}, ..., s^0, a^0) = P(a^t = a|s^t) \quad (1)$$

which means that a state transitions only on the previous state and the current action. Thus, only the last state and action are interesting regarding the decision for the next state.

In a Markov Decision Process, the agents are trying to find a policy which maximizes the sum of discounted expected rewards. The standard solution for this is through iterative search method which searches for a fized point of the *Bellman equation*:

$$v(s, \pi^*) = max_a(r(s, a) + \gamma \sum_{s'} p(s'|s, a)v(s', \pi^*)) \quad (2)$$

B. Reinforcement Learning

When the state transition probabilities or the rewards are unknown, the problem of the Markov Decision Process becomes a problem of Reinforcement learning. In this group of problem the agent tries to make a model of the world around itself by trial and error.

One type of reinforcement learning is **value-based reinforcement learning.**In this case, the agent tries to learn a value function that renders a value to the states or to the actions from states. These values correspond to the achievable reward from reaching a state or from taking a specific action from a state.

The most commonly used type of value-based reinforcement learning is **Q-learning**, when the so-called Q-values are estimated for each of the state-action pairs of the world. These Q-values represent the value of choosing a specific action in a state, meaning how much reward could the agent possibly get by taking that action. The equation for Q-learning for updating the Q-values of a state is:

$$Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a')) \quad (3)$$

where α is the learning rate and γ is the discount for the reward.The agent always selects an action that maximizes the Q-function for the state that the agent is in.

Another type of reinforcement learning is **policy-based reinforcement learning**. In this case, actions are derived as a function of the state itself. The most common policy-based reinforcement learning method is **policy gradient**. In this case, the agent tries to maximize the expected reward following the policy π_θ parametrized by θ, based on the total reward for a given trajectory $r(\tau)$. Thus, the cost function of the parameters θ is the following:

$$J(\theta) = E_{\pi_\theta}[r(\tau)] \quad (4)$$

The parameters are then tuned based on the gradient of the cost function:

$$\theta_{k+1]} = \theta_t + \alpha \Delta J(\theta_t) \quad (5)$$

An advantage of using policy-based methods is the possibility of mapping environments with huge, even continuous action spaces. Environments with stochasticity can also be solved with them. However, it comes with the disadvantage of the much greater possibility of getting stuck in a local maximum rather than following the optimal policy.

Apart from the aforementioned model-free reinforcement learning methods, there exists also **model-based reinforcement learning**. In this case, a model is built(or just tuned) to perform the reinforcement learning. This is more sample-efficient than model-free methods, thus it requires less samples to perform equally, but it is very dependent on the model built. It can be combined with model-free methods to achieved better results, as in [19].

C. Multi-agent systems, Markov games

A matrix game is a stochastic framework where each player selects an action and gets their immediate reward based on its and all other agents' action. They are called as matrix games due to the fact that the games can be written as a matrix, with the first two player selecting action in the row and the column of the matrix. Unlike Markov Decision Processes, these games have no state.

Markov games, or Stochastic games are an extension of Markov Decision Processes with multiple agents. Also, it can be thought of as an extension to Matrix games with multiple states. In a Markov game, each state has a payoff matrix for all of the states. The next state is determined by the joint action of the agents. A game is Markovian if

$$P(a_i^t = a_i | s^t, a_i^{t-1}, ..., s^0, a_i^0) = P(a_i^t = a_i | s^t) \quad (6)$$

so the next state depends only on the current state and the current actions taken by all agents.

D. Deep Reinforcement Learning

A reinforcement learning algorithm is called deep reinforcement learning algorithm if it is assisted by a neural network.

A neural network is a function approximator built from (even billions and billions of) artificial neurons. An artificial neuron, based on real neurons of the brain, has the following equation:

$$y = Act(\sum wx + b) \quad (7)$$

where x is the input vector, w is the weight vector, b is the bias and $Act()$ is the activation function to introduce nonlinearity in an otherwise linear system. The parameters(w and b) are tuned with backpropagation, calculating the partial derivative error of all parameters propagated from the final error up until the input vector.

The selection of the activation function is important in deep learning due to the vanishing gradients: when many layers are stacked upon each other, higher layers' gradients are too small during backpropagation, thus those layers are difficult to train. A basic activation function can be a sigmoid or logistic activation function:

$$y = \frac{1}{1 + e^{-x}} \quad (8)$$

A common choice of activation function in deep learning is Rectified Linear Unit (ReLU) [20] which has gradients that are less vanishing, thus better to train. It has the following equation:

$$\begin{aligned} y = x \text{ if } x > 0 \\ y = 0 \text{ if } x <= 0 \end{aligned} \quad (9)$$

For multi-class classification, another activation function is used: the softmax activation function. When used as the last layer, the probabilities of all of the output neurons add up to exactly 1. Thus, in reinforcement learning it is utile to use it as the probability distribution of the possible actions. It has the following equation:

$$y = \frac{e^{x_i}}{\sum_j e^{y_j}} \quad (10)$$

Deep reinforcement learning algorithms have several advantages compared to traditional reinforcement learning algorithms. First of all, they are not based on a state table anymore, as the states are approximated(much more robust than linear function approximators). This allows much more states to be mapped, or even allow the states to be continuous. However, it is more prone to diverging and thus many optimizations have been created on deep reinforcement learning algorithms to provide better convergence on the problems.

E. Actor-critic

An Actor-critic system is the combination of value-based and policy-based reinforcement learning. In these systems there are two distinct parametrized networks: the Critic, which estimates a value function (like in value-based reinforcement learning), and an Actor, which updates the policy network based on the direction suggested by the Critic (like in policy-based reinforcement learning). Actor-critic algorithms follow an approximate policy gradient:

$$\begin{aligned} \nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) Q_w(s,a) \\ \Delta\theta = \alpha \nabla_\theta log \pi_\theta(s,a) Q_w(s,a) \end{aligned} \quad (11)$$

Approximating the policy gradient introduces bias to the system. A biased policy gradient may not find the right solution, but if we choose value function approximation carefully, then we can avoid introducing any bias.

Actor-critic systems generally perform better than regular reinforcement learning algorithms. The critic network enables that the system does not get stuck in a local maximum, meanwhile the Actor network lets the mapping of environments with huge action spaces as well as providing better convergence.

F. A2C algorithm

A2C is the abbreviation of Synchronous Advantage Actor-Critic. It is a one-environment-at-a-time derivation of A3C(Asynchronous Advantage Actor-Critic) algorithm [21], which processed multiple agent-environments simultaneously. In that, multiple "workers" update a global value function, thus exploring the state space effectively. However, the Synchronous Advantage Actor-Critic provides better performance compared to the asynchronous model.

Advantage function is a method to significantly reduce the variance of the policy gradient by substracting the cumulative reward with a baseline to make smaller gradients, thus it provides much better convergence than regular Q-values.

$$A(s,a) = Q(s,a) - V(s) \quad (12)$$

Returns are calculated as the following equation:

$$G_t = r_t + \gamma * r_{t+1} * (1 - T_t) \quad (13)$$

where G is the return, r_t is the reward at time t, γ is the discount factor and T_t is whether the step at time t is a terminal state.

III. Experiments and Results

Figure 2 shows the simulation environment of our work. The testbed is an 5x5 grid, where are three cooperating agents in three of the four corners of the environment(the squares). In the middle, there is a fourth agent(the circle). The former three agents have the objective of catching the fourth agent, which moves randomly. The agents can move in four directions: up, down, left or right. When one of the three agents catch

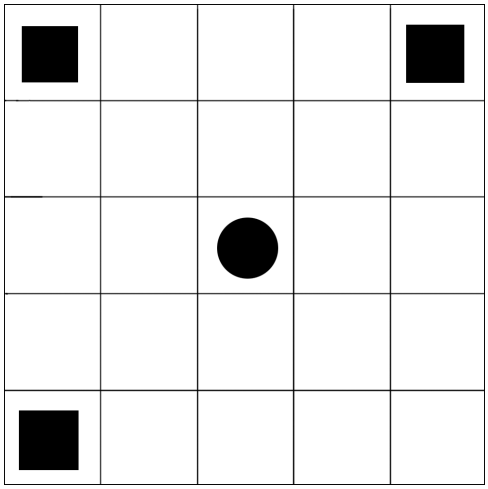

Fig. 2. Figure about the simulation environment. The squares the controlled agents, meanwhile the circle represents the fleeing enemy. The goal is to catch the enemy by moving horizontally or vertically.

the fourth one, the episode is ended. A penalty is introduced to the cooperative agents every timestep, thus the return of an episode is maximized by ending the episode as soon as possible (by catching the fleeing agent as fast as they can). and each episode must end in 1000 timesteps to avoid getting stuck in an episode. In the environment, a constant negative reward is given to the agents which encourages them to end the episode as fast as they can.

In the modification of the A2C algorithm, we followed the theory of centralized learning and decentralized execution. This means that it is enough to have the execution part decentralized, the learning phase can be assisted by additional information from other agents. In our case, we used the information that the agents are cooperative, thus they have acquire the same rewards (and returns, as well).

In our experiment, the multitudes of A2C models of one actor and one critic was substituted by one model of one critic and multiple actors. All neural network layers were as a subclass of the TensorFlow Model class, which provides utile functions for training and prediction, even for batch tasks by only providing the forward steps of the network. The optimizer was chosen to be RMSprop with learning rate of 7e-3.

The value estimator critic contained a neural network 128 hidden unit layers with ReLU activation function and one output layer with one unit. Its loss function is a simple mean squared error between the returns and the value.

The actors contained a hidden layer with 128 hidden units and an output layer with 4 units (being equal to the number of the actions in the action space). The loss function contained two distinct parts: policy and entropy loss. The policy loss was a weighted sparse categorical cross-entropy loss, where the weights are given by the advantages. This method increases the convergence of the algorithm. Entropy loss is a method to increase exploration, by tending to take actions that are not in the local minimum. This is very important for tasks with sparse reward due to the fact that the agent does not receive feedback often. This loss is calculated as a cross-entropy over itself, and it is substracted from the policy loss because it should be maximized, not minimized. The entropy loss is tuned by a constant, which is taken as 1e-4.

Episode rewards were taken to be a list where a value of 0 was appended to the end of the list at each episode end, and during the episodes, only the last value of the list was incremented by the episode reward of the given step. For the training, a batch-sized container is created for the actions, rewards, terminal state booleans, state values and observed states. Then, a two-level loop is started: the outer one is run for the number of required updates(set by us), meanwhile the inner loop size is equal to the batch size. In the inner loop, the environment is run for batch size times and the state observations, the taken actions (selected by a probability distribution based on the actor neural network results), the state values, the rewards, the terminal state booleans and the last observed state are stored in the aforementioned containers. Then, with the collected data, the returns and the advantages are calculated on the batch, and then a batch training is performed on the collected data. There was no need to calculate the gradients themselves due to the usage of the Keras API.

During our experiment, the system was run 5000 times in batches of 128, thus running the environments over totally 640000 steps. Gamma was taken to be 0.99.

Figure 5, Figure 6 show the results of our experiments. The most important is to check the x coordinates of Figure 5 and Figure 6: for the same number of steps, the original was run for 40340 episodes meanwhile the modified algorithm managed to complete 82119 episodes. This means that the A2CM algorithm spent half as much steps in an episode, thus it was able to catch the fleeing opponent in average in half time than the agent based on the original algorithm. These figures also show that the original algorithm did not find an optimal solution without diverging later, and even between diverges the solutions were not as stable. Our agent, on the other hand, has found a solution with no diverges later and only small diverges after the first half of episodes. The A2CM algorithm has found a solution where it can catch the opponent in 6 steps, and it had maintained the knowledge for 20000 episodes, with one positive spike where it found the solution to the problem in just 3 steps.

Run times are worth to take to consideration as well. The regular A2C algorithm took 14567.45 seconds to run, meanwhile the modified one ran for 14458.28 seconds. It is worth noting that due to the almost twice more episodes, the environment had to be reset twice as more, so the modified algorithm is even faster that the normal one.

IV. Conclusion

As it was seen in the previous section, our modification on the original A2C algorithm, the A2CM algorithm was able to perform much better on our testbed than the original one.Although, the algorithm has the caveat of only being able to be used when the agents are fully cooperative without any special predefined roles between them.

Algorithm 1 A2CM

Initialize Model:
Initialize N+1 hidden and N+1 output (1 value + N action) layers (4 different networks in one model, 1 critic + 3 actor)
for number of updates **do**
for batch size **do**
Calculate next actions a based on the previous state
Take actions a, get terminal state boolean and new rewards
Store the actions, the terminal state booleans, the calculated values, the rewards and the states
end for
Calculate returns based on (13)
Calculate advantages based on (12)
Update critic neural network based on the observed states and the corresponding returns
loss is MSE between returns and calculated values
Update actor neural networks based on the observed states, the taken actions and the advantages
loss is policy loss(weighted sparse categorical cross-entropy loss) - entropy loss(cross-entropy over itself)
end for

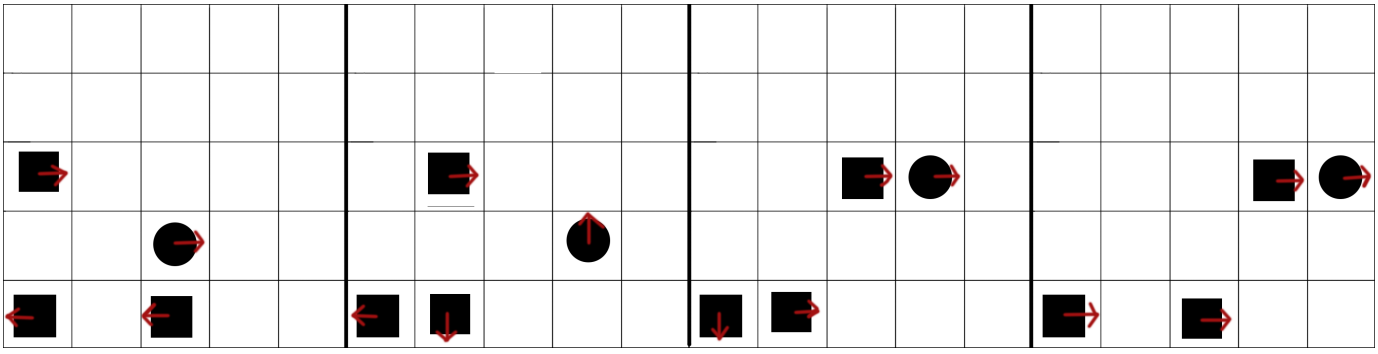

Fig. 3. An example of catching the randomly moving opponent.

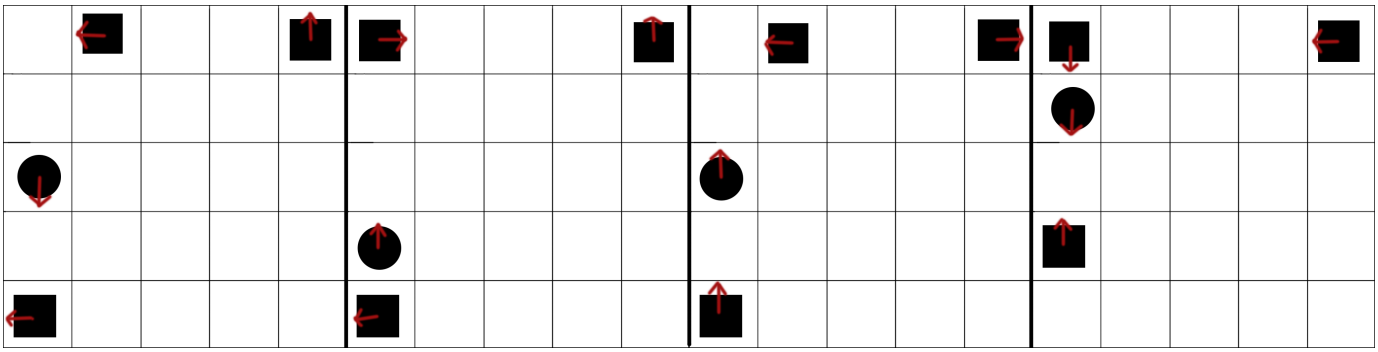

Fig. 4. An example of catching the fleeing opponent.

However, there are still a lot of possibilities to improve upon the current state of our algorithm. First of all, it can be examined how this algorithm would behave when collision is tracked between the agents. Collision avoidance is an important factor in robotics and is rarely checked in regular pursuit-evasion environments. Also, another possibility of improvement would be to introduce a variable learning rate like Win or Learn Fast [3] in a deep reinforcement learning algorithm. Another point of possible improvement is the inclusion of the fleeing agent in the algorithm to have it cope with the full cooperative-competitive being of the environment. Also, another activation functions can be tried to check their behavior, for example, Exponential Linear Units [22] might have better convergence for a price of slightly more training time. An extension to this algorithm could be the introduction of the possibility of using recurrent neural networks, to be able to deal with Partially Observable Markov Decision Processes (POMDPs) where the full state is unknown.

ACKNOWLEDGEMENT

The research reported in this paper and carried out at the Budapest University of Technology and Economics was sup-

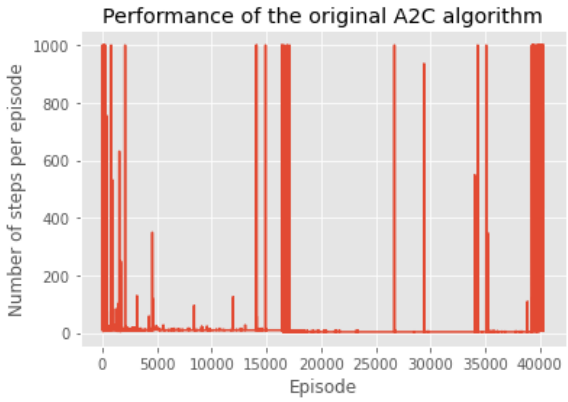

Fig. 5. Figure about the performance of the original A2C algorithm on our benchmark.

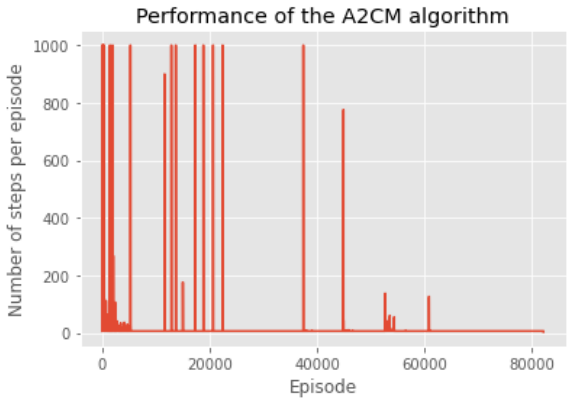

Fig. 6. Figure about the performance of the modified A2C algorithm on our benchmark.

ported by the "TKP2020, Institutional Excellence Program" of the National Research Development and Innovation Office in the field of Artificial Intelligence (BME IE-MI-SC TKP2020).

REFERENCES

[1] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *In Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann, 1994, pp. 157–163.

[2] J. Hu and M. Wellman, "Nash q-learning for general-sum stochastic games." *Journal of Machine Learning Research*, vol. 4, pp. 1039–1069, 01 2003.

[3] M. Bowling and M. Veloso, "Multiagent learning using a variable learning rate," *Journal = "Artificial Intelligence*, no. 136, pp. 215–250, 2002.

[4] M. H. Bowling and M. M. Veloso, "Simultaneous adversarial multi-robot learning," in *IJCAI*, 2003, pp. 699–704.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning."

[6] J. Foerster, N. Nardelli, G. Farquhar, T. Afouras, P. H. S. Torr, P. Kohli, and S. Whiteson, "Stabilising experience replay for deep multi-agent reinforcement learning."

[7] S. Omidshafiei, J. Pazis, C. Amato, J. P. How, and J. Vian, "Deep decentralized multi-task multi-agent reinforcement learning under partial observability."

[8] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson, "Counterfactual multi-agent policy gradients."

[9] P. Peng, Y. Wen, Y. Yang, Q. Yuan, Z. Tang, H. Long, and J. Wang, "Multiagent bidirectionally-coordinated nets: Emergence of human-level coordination in learning to play starcraft combat games."

[10] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls, and T. Graepel, "Value-decomposition networks for cooperative multi-agent learning."

[11] T. Rashid, M. Samvelyan, C. S. de Witt, G. Farquhar, J. Foerster, and S. Whiteson, "Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning."

[12] R. Lowe, Y. WU, A. Tamar, J. Harb, O. Pieter Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 6379–6390. [Online]. Available: http://papers.nips.cc/paper/7217-multi-agent-actor-critic-for-mixed-cooperative-competitive-environments.pdf

[13] Shihui, Y. Wu, X. Cui, H. Dong, Fang, Fei, and S. Russell, "Robust multi-agent reinforcement learning via minimax deep deterministic policy gradient," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, Jul. 2019, pp. 4213–4220.

[14] P. Casgrain, B. Ning, and S. Jaimungal, "Deep q-learning for nash equilibria: Nash-dqn."

[15] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "Starcraft ii: A new challenge for reinforcement learning."

[16] M. Samvelyan, T. Rashid, C. S. de Witt, G. Farquhar, N. Nardelli, T. G. J. Rudner, C.-M. Hung, P. H. S. Torr, J. Foerster, and S. Whiteson, "The starcraft multi-agent challenge."

[17] S. Liu, G. Lever, J. Merel, S. Tunyasuvunakool, N. Heess, and T. Graepel, "Emergent coordination through competition."

[18] N. Bard, J. N. Foerster, S. Chandar, N. Burch, M. Lanctot, H. F. Song, E. Parisotto, V. Dumoulin, S. Moitra, E. Hughes, I. Dunning, S. Mourad, H. Larochelle, M. G. Bellemare, and M. Bowling, "The hanabi challenge: A new frontier for ai research."

[19] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine, "Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning."

[20] A. F. Agarap, "Deep learning using rectified linear units (relu)."

[21] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning."

[22] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)."